Managing Plugin Load Orders in Video Game Modding Using Directed Acyclic Graphs: A Case Study of Skyrim

Faishal Ahmad Nurdin - 13525027
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: fanoflegend@gmail.com, 13525027@std.stei.itb.ac.id

***Abstract*—Modding is one of the key components that extend a video game longevity. It offers more variation of the game to the player that can keep them engaged for a long time. One of the video games famous for its mods is *The Elder Scrolls V: Skyrim*. However, when players install a large collection of mods, they need to integrate increasingly large collections of user-created plugins. Because of this, managing the "load order"—the sequential execution of these modifications—becomes a complex and error-prone challenge. Improper load orders frequently result in unintentional data overwrites, localized bugs, and catastrophic application crashes. This paper proposes a robust, graph-theoretic approach to automate and optimize plugin management using Directed Acyclic Graphs (DAGs). By mapping individual plugins as vertices and their inter-dependencies, prerequisites, and conflict-resolution rules as directed edges, we mathematically model the modding environment. Applying topological sorting algorithms to this DAG ensures that all master files and requisite modifications are loaded in a computationally stable sequence.**

Keywords: Video Game Modding, Directed Acyclic Graphs (DAG), Dependency Management, *The Elder Scrolls V: Skyrim*, Load Order Optimization.

I. INTRODUCTION

Over the past two decades, video game modding—the practice of players modifying game software to alter gameplay, graphics, or content—has evolved from a niche hobby into a cornerstone of the interactive entertainment industry. User-generated content such as modding not only normalized game development but also highly increases the commercial lifespan and player retention of major software titles. An example of this phenomenon is Bethesda Game Studios' The Elder Scrolls V: Skyrim (2011). Skyrim has tens of thousands of user-created modifications (mods) that players can use for free available on websites such as Nexus Mods and Steam Workshop. However, this flexibility introduces significant technical complexities, primarily centralized around how the game engine handles external data integration.

In Skyrim's game engine, the Creation Engine, modifications are packaged as discrete plugins. Because multiple plugins frequently attempt to modify the same base game data—such asI altering the attributes of a single non-player character (NPC) or modifying the geometry of a specific landscape—the engine relies on a strict sequential loading mechanism known as the "load order." Under this paradigm, conflicts are resolved via the "Rule of One," where the plugin loaded last overwrites any conflicting data from previously loaded plugins.

As end-users increasingly curate massive collections containing hundreds or even thousands of active plugins, managing this sequence manually becomes computationally intractable for a human operator. An incorrect load order results in broken dependencies, missing assets, logic errors, and severe application crashes (frequently referred to as CTDs, or "Crashes to Desktop"). While community-driven heuristics and manual conflict-resolution patches exist, they lack a formalized mathematical framework to guarantee stability and prevent circular logic errors.

To address the inherent fragility of manual plugin sequencing, this paper proposes framing load order management as a graph-theoretic dependency problem. By abstracting the modding environment into a Directed Acyclic Graph (DAG), this can rigorously and automatically resolve plugin sequences. In this graph model Vertices (Nodes) represent individual game plugins and Directed Edges represent the categorical rules and dependencies between plugins. A valid load sequence cannot contain infinite loading loops called circular dependencies (e.g. A requires B, but B requires A). Because of that, the underlying data structure must remain acyclic and such Directed Acyclic Graph (DAG) is used. By constructing this DAG, topological sorting algorithms can be applied to instantly generate a mathematically valid, conflict-free load sequence, while simultaneously flagging cyclic errors.

II. THEORETICAL FRAMEWORK

A. Graph

Graph is a diagram or data structure that can be used to represent complex, non-linear relationships between discrete objects. Graph is a collection of points, known as vertices (nodes), and lines, known as edges (arcs), connecting some (possibly empty) subset of them.

The study of graphs is known as Graph Theory and first systematically studied by D. König in the 1930s. Graph can be defined as G = (V, E). V stands for non-empty set of vertices and E stands for set of edges (can be empty). Graphs can be classified in many ways, such as based on orientation, weight, structure, cycles, and connectivity.

Based on orientation, graphs are divided into two types, undirected graph dan directed graph.

1. An undirected graph is a graph whose edges have no orientation. If two vertices are connected by an edge, the connection goes both ways. It means you can travel in either direction (from A to B or from B to A).

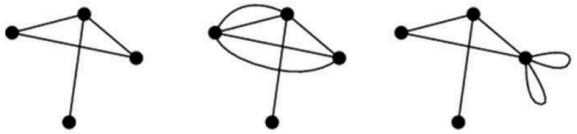


Fig. 1. Undirected Graphs
(Source:https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

2. A directed graph is a graph whose edges have orientation. Edges that connect two vertices have arrows indicating the orientation of the traversal. It means you can only travel to the direction of the arrows.

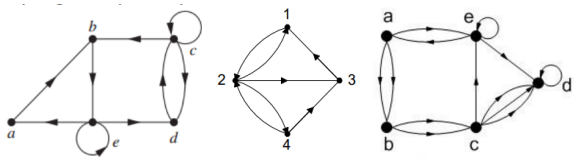


Fig. 2. Directed Graphs
(Source:https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf)

In graph theory, a cycle is a non-empty, closed path where the sequence of vertices starts and ends at the same node, and no edges or intermediate vertices are repeated. Based on the presence of cycles, graphs are divided into two types, cyclic graph and acyclic graph.

1. A cyclic graph is a graph that contains at least one cycle.

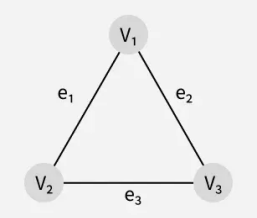


Fig. 3. Cyclic Graph
(Source:https://www.geeksforgeeks.org/dsa/graph-types-and-applications/)

Based on the figure above, vertices V1, V2, and V3 are connected in such a way that they form a closed loop. Because of that, the graph is a cyclic graph.

2. An acyclic graph is a graph that contains no cycles. One of the common examples of acyclic graphs is trees. You can't form a cycle in trees because trees do not contain loops that form a cycle.

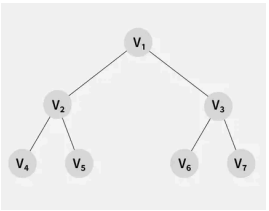


Fig. 4. Acyclic Graph
(Source:https://www.geeksforgeeks.org/dsa/graph-types-and-applications/)

B. Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a directed graph that is also an acyclic graph. DAG has the combined properties of a directed graph and an acyclic graph, thus it has orientation (indicated by arrows) and does not contain cycles. Orientation indicates a one-way relationship and does not contain cycles means that you cannot traverse a sequence of directed edges and return to the same node.

Directed acyclic graphs have many similarities with trees, but they are different. The main difference is that a child node in trees can only have one parent node, while DAGs allow a child node to have more than one parent. Because of this, all trees are DAGs, but not all DAGs are trees.

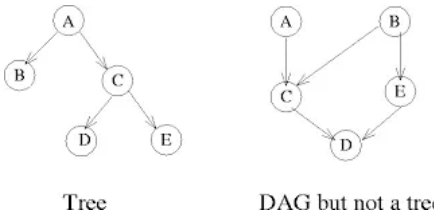


Fig. 5. Tree and DAG
(Source:https://tarunjain07.medium.com/dag-tree-5cc71c2c0c85)

DAG has some properties that are useful to solve graph related problems. They are:

1. Reachability Relation: Using DAG, the existence of a reachability relation between two nodes can be determined. If you can follow the direction of edges in the graph to get from B to A, thus there is a direct path that starts at node B and ends at node A, it is implied that node A is reachable from node B.
2. Transitive Closure: The transitive closure of a directed graph is a new graph that represents all the direct and indirect relationships or connections between nodes in the original graph. Transitive closure tells you which nodes can be reached from other nodes by following one or more directed edges.

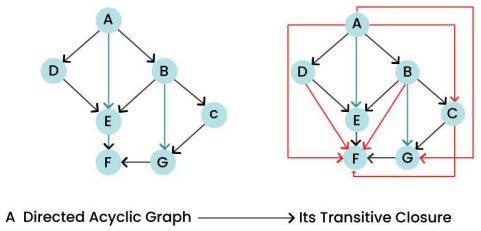


Fig. 6. Transitive Closure

(Source:https://www.geeksforgeeks.org/dsa/introduction-to-directed-acyclic-graph/)

3. Transitive Reduction: The transitive reduction of a directed graph is a new graph that retains only the essential, direct relationships between nodes. It removes any unnecessary indirect edges that can be inferred from the other edges, thus simplifying the graph.

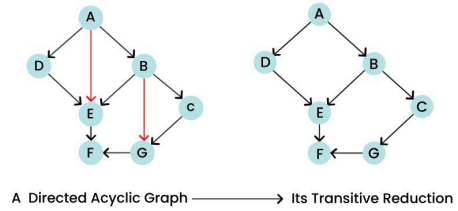


Fig. 7. Transitive Reduction
(Source:https://www.geeksforgeeks.org/dsa/introduction-to-directed-acyclic-graph/)

4. Topological Ordering: The topological ordering of a directed acyclic graph is a topological sorting, in which you linearly order its nodes in such a way that for all the edges, the start node of the edge occurs earlier in the sequence. This DAG property can be used for scheduling and dependency resolution.

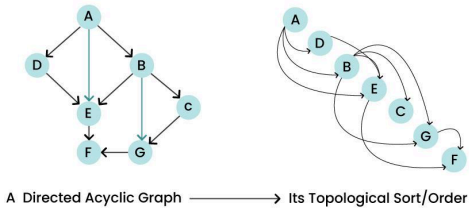


Fig. 8. Topological Ordering
(Source:https://www.geeksforgeeks.org/dsa/introduction-to-directed-acyclic-graph/)

C. Kahn's Algorithm

Kahn's Algorithm is a highly efficient method used in computer science to find a topological sort of a Directed Acyclic Graph (DAG). It was created by Arthur B. Kahn in 1962. Kahn's Algorithm operates with a linear time complexity of O(V + E), making it efficient for processing load orders containing hundreds of plugins. Kahn's Algorithm works by following a set of steps:

1. Count In-Degrees: Calculate the in-degree for every single node in the graph. In-degree is the number of arrows pointing into a node.
2. Find the Starters: Find all nodes that have an in-degree of 0 and put them in a queue.
3. Process the Queue: Take a node out of the queue and add it to your final "Sorted List". Look at all the arrows pointing out of that node to other nodes. "Delete" those arrows. By deleting them, you subtract 1 from the in-degree of the destination nodes. Repeat this until any of those destination nodes now drop to an in-degree of 0, then immediately add them to the queue. Keep doing this until the queue is empty. If you sorted all the nodes, you have a perfect load order. If you still have nodes left over but the queue is empty, you have a cycle.

```cpp
vector<int> topoSort(vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> indegree(n, 0);
    queue<int> q;
    vector<int> list;

    // Compute indegrees
    for (int i = 0; i < n; i++) {
        for (int next : adj[i])
            indegree[next]++;
    }

    // Add all nodes with indegree 0
    // into the queue
    for (int i = 0; i < n; i++)
        if (indegree[i] == 0)
            q.push(i);

    // Kahn's Algorithm (BFS)
    while (!q.empty()) {
        int top = q.front();
        q.pop();
        list.push_back(top);
        for (int next : adj[top]) {
            indegree[next]--;
            if (indegree[next] == 0)
                q.push(next);
        }
    }

    return list;
}
```

Fig. 9. Kahn's Algorithm C++ Code
(Source:https://www.geeksforgeeks.org/dsa/topological-sorting-indegree-based-solution/)

D. Modding

A game mod describes a modification within an existing commercial video game that has been created by users or independent creators. The word mod is short for modification. A mod can be a change to any part of a game, including how it sounds, how it plays, the way it looks, or anything else. The size of mods can be as small or as large as the creators want it to be. They can be as small as minor bug fixes, or as big as creating a completely new game on the existing game.

Modding is playing a crucial part in extending a game lifespan. It brings new varieties to the game so players can enjoy a different experience when playing a game that they already finish. Because of mods, there are many players that still play games that were released more than a decade ago. There are many games that are famous for their modding community, such as Skyrim, Fallout 4, Minecraft, Terraria, and many more.

Mod works by altering a video game's code, asset, or logic. Different types of mods change different aspects of a game. Generally mods work through three method:

1. Asset Replacement: Mods overwrite original game files. For example, swapping an original texture file with a higher-resolution version to improve graphics.
2. Data-Driven Configuration: Many games are designed to be read easily by third parties. Modders edit readable configuration files (like XML, JSON, or Lua) to adjust things like item drop rates, character speed, or menu layouts.
3. Script & Code Injection: Advanced mods add entirely new mechanics or systems. They inject custom scripts that the game engine reads alongside its base files to trigger new events.

E. Modding of Skyrim

To mod a game, there are rules that must be met. These rules can be different for one game to the other. Skyrim game engine, the Creation Engine, uses The "Rule of One" to resolve data conflicts between multiple active plugins. Because the engine cannot natively merge conflicting data changes at runtime, it relies on a strict overwrite system dictated by the load order.

When the game launches. The engine reads through the active load order sequentially, from top to bottom. Every item, character, weather pattern, and location in the game exists as a specific "record" with a unique FormID. If two or more plugins attempt to modify the exact same base record, the engine applies the Rule of One: the plugin that is loaded last (the lowest in the load order) completely overwrites all changes made to that specific record by any previously loaded plugins. The engine only recognizes one "winning" version of a record. So only one rule can be applied to a base record, the other will be ignored by the game engine.

Skyrim mods consist of one or more files or components types. These files have different functions and use cases.

1. Plugin Files (The Instructions)

   These files are the core of most mods. They are built using the Creation Kit (or community tools like xEdit) and tell the game engine exactly what data to load, modify, or inject.
   - .esm (Elder Scrolls Master): These are the foundational files. The base game and official DLCs are .esm files (e.g., Skyrim.esm). Modders use them for massive, structural mods (like new lands or major overhauls) because the engine loads them first, providing a base for other plugins to build upon.
   - .esp (Elder Scrolls Plugin): The most common mod file. These are loaded after the master files and are used for modifying everything, from adding a single NPC to changing lighting values. If an .esp modifies something an .esm or a previously loaded .esp already touched, the "Rule of One" dictates that the last loaded .esp wins out.
   - .esl (Elder Scrolls Light): These are the "lightweight" plugins. These plugins work similar to the .esm and .esp but have much bigger limits. .esm/.esp files have an active limit of 255 plugins, but .esl files have an active limit of over 4,000 plugins.
2. Asset Archives (The Packaging)

   .bsa (Bethesda Softworks Archive): It functions very much like a .zip or .rar file. The game engine efficiently reads directly from these archives to pull the necessary assets when a corresponding .esp or .esm asks for them.
3. Loose Files (The Raw Materials)

   If a mod author doesn't pack their assets into a .bsa, you will see folders like Meshes, Textures, or Sound. These are "loose files." The game engine will always prioritize loading a loose file over an asset packed inside a .bsa.
   - .dds (DirectDraw Surface): The standard file format for 2D textures.
   - .nif (NetImmerse File): The 3D models/meshes of objects, characters, and architecture.
   - .pex (Compiled Papyrus Script): The code that runs the logic in the game (e.g., triggering a quest stage when you read a book). .psc files are the uncompiled, human-readable source code for these scripts.
   - .xwm or .wav: Audio files for voice acting, sound effects, and music.
4. Engine Extensions (The Power Users)

   Mods that go beyond what the base Creation Engine allows rely on the Skyrim Script Extender (SKSE). These mods interact directly with the game's memory.
   - .dll (Dynamic Link Library): These are essentially custom-coded programs (usually written in C++) that inject directly into the game engine via SKSE. They allow for massive mechanical changes, complex UI overhauls (like SkyUI), or advanced combat behaviors that the base game simply cannot handle.
   - .ini or .json: Configuration files. Mods that use .dll files or complex scripts often generate these so you can customize the mod's settings.

III. METHODOLOGY

A. Mathematical Formulation of the Modding Environment

To resolve plugins conflict, the modding environment will be modeled as a directed graph. This directed graph is defined as $G = (V, E)$, where:

- V (Vertices): Represent the set of all plugins (.esm, .esp, .esl) present in the user's directory.
- E (Edges): Represent the set of direct dependencies relation between plugins. A directed edge $(u, v) \in E$ indicates that plugin u must be loaded before plugin v.

To ensure load working and generate the desirable outcome, there are two rules implemented in this directed graph model:

1. If plugin v has any dependencies to master plugin u, the relationship is absolute. Failure to satisfy (u, v) results in a critical runtime crash.
2. If plugins u and v modify the same base record, but the user intends for v's changes to take precedence, a "load-after" rule generates edge (u, v).

For the game engine to successfully initialize all plugins, circular dependencies must not occur, thus a cycle must not be formed in graph G. Because of that, graph G must be a directed acyclic graph (DAG).

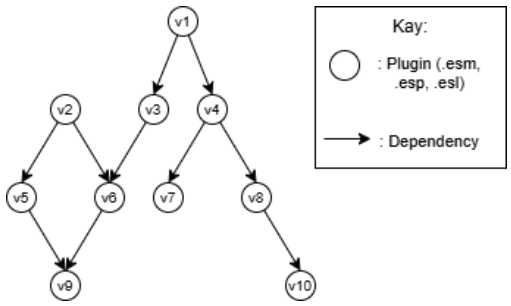


Fig. 10. Graph Model for Mathematical Formulation
(Source: Author)

B. Cycle Detection and Topological Sorting

To generate the final load order sequence, a topological sorting algorithm is applied to graph G. A valid topological sort is a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering.

Our implementation utilizes Kahn's Algorithm. The process initializes by identifying the subset of vertices with an in-degree of 0 (base game master files such as Skyrim.esm, which have no prerequisites).The algorithm proceeds as follows:

1. Initialize an empty list L to contain the sorted elements.
2. Initialize a queue S containing all vertices with no incoming edges (in-degree = 0).
3. While S is not empty:
   - Remove a vertex n from S and append it to L.
   - For each node m with an edge e from n to m:
     - Remove edge e from the graph.
     - If m has no other incoming edges, insert m into S.

If, upon termination, the graph still contains edges, it indicates that at least one cycle exists, meaning a topological sort is impossible. This indicates there are circular dependencies in the mod list, thus an error occurs. In this framework, this will trigger an error-handling routine that isolates the cyclic vertices and outputs the conflicting loop to the user for manual resolution.

It is important to note that the topological sorting algorithm does not actively resolve record-level data conflicts or execute the 'Rule of One'. The Rule of One is an inherent overwrite mechanism of the Creation Engine. Instead, the DAG framework serves as a conflict-management proxy. By defining a directed edge (u, v), the user explicitly instructs the algorithm to sequence vertex v after vertex u. This guarantees that when the resulting load order is passed to the game engine, the engine's native Rule of One will predictably resolve the data conflict in favor of plugin v without causing runtime instability.

IV. SYSTEM IMPLEMENTATION

A. Data Structure

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_PLUGINS 100
#define MAX_NAME_LEN 50

typedef struct Node {
    int dest;
    struct Node* next;
} Node;

typedef struct Graph {
    int num_plugins;
    char plugin_names[MAX_PLUGINS][MAX_NAME_LEN];
    Node* adj_list[MAX_PLUGINS];
    int in_degree[MAX_PLUGINS];
} Graph;
```

Fig. 11. Program Setup
(Source: Author)

This section defines the foundational architecture of the system. For this program, the system utilizes an adjacency list. The "Node" structure acts as a linked list element representing a directed edge (a dependency). The "Graph" structure encapsulates the entire modding environment. It also includes the "in_degree" integer array, which tracks exactly how many prerequisites each plugin has. This is the core metric required for Kahn's Algorithm.

B. System Initialization

```c
Node* createNode(int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}

Graph* createGraph(int num_plugins) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->num_plugins = num_plugins;

    for (int i = 0; i < num_plugins; i++) {
        graph->adj_list[i] = NULL;
        graph->in_degree[i] = 0;
    }
    return graph;
}

void setPluginName(Graph* graph, int id, const char* name) {
    if (id >= 0 && id < graph->num_plugins) {
        strncpy(graph->plugin_names[id], name, MAX_NAME_LEN - 1);
        graph->plugin_names[id][MAX_NAME_LEN - 1] = '\0';
    }
}
```

Fig. 12. System Initialization
(Source: Author)

These functions are used to allocate memory and initialize the starting state of the program before any sorting occurs. "createGraph" dynamically allocates the necessary memory based on the total number of mods (num_plugins) input by the user, explicitly setting all initial in_degree values to 0 and emptying the adjacency lists. The "setPluginName" function acts as the data ingestion layer, mapping human-readable string identifiers (such as "Skyrim.esm") to the numerical vertex IDs used by the algorithm.

C. Dependency Mapping

```c
void addDependency(Graph* graph, int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph->adj_list[src];
    graph->adj_list[src] = newNode;
    graph->in_degree[dest]++;
}
```

Fig. 13. Dependency Mapping
(Source: Author)

This function is responsible for building the topological constraints of the load order. When a user or a metadata file dictates that a source plugin (src) must load before a destination plugin (dest), this function creates a directed edge. It performs two critical actions: first, it prepends the destination node to the source node's adjacency list. Second, and most importantly, it increments the in_degree of the destination node by 1, mathematically registering that the destination mod now has an unfulfilled prerequisite.

D. Kahn's Algorithm

```
void calculateLoadOrder(Graph *graph)
{
    int queue[MAX_PLUGINS];
    int front = 0, rear = 0;
    int sorted_order[MAX_PLUGINS];
    int processed_count = 0;
    for (int i = 0; i < graph->num_plugins; i++)
    {
        if (graph->in_degree[i] == 0)
        {
            queue[rear++] = i;
        }
    }
    while (front < rear)
    {
        int current_plugin = queue[front++];
        sorted_order[processed_count++] = current_plugin;
        Node *temp = graph->adj_list[current_plugin];
        while (temp != NULL)
        {
            int neighbor = temp->dest;
            graph->in_degree[neighbor]--;
            if (graph->in_degree[neighbor] == 0)
            {
                queue[rear++] = neighbor;
            }
            temp = temp->next;
        }
    }
```

Fig. 14. Kahn's Algorithm
(Source: Author)

This part of the program executes Kahn's Algorithm using a First-In-First-Out (FIFO) integer queue. First, it scans the environment and queues any plugin with an in_degree of 0 (indicating it has no dependencies). After that, a while loop systematically dequeues these independent plugins, adding them to the final sorted_order array. As each plugin is processed, the system "deletes" its outgoing edges by decrementing the in_degree of its neighbors. If a neighbor's in_degree drops to 0, its prerequisites have been fulfilled, and it is pushed into the queue.

E. Cycle Detection

```
    if (processed_count != graph->num_plugins)
    {
        printf("\nERROR! Circular Dependency Detected!\n");
        printf("The graph is not a valid DAG. A topological sort is impossible.\n");
        printf("The following plugins are caught in a circular loading loop:\n");
        for (int i = 0; i < graph->num_plugins; i++)
        {
            if (graph->in_degree[i] != 0)
            {
                printf(" - %s\n", graph->plugin_names[i]);
            }
        }
    }
    else
    {
        printf("\nSUCCESS! Valid Load Order Generated:\n");
        printf("---------------------------------------\n");
        for (int i = 0; i < processed_count; i++)
        {
            printf("%02d. %s\n", i + 1, graph->plugin_names[sorted_order[i]]);
        }
        printf("---------------------------------------\n");
    }
}
```

Fig. 15. Cycle Detection
(Source: Author)

This logic is used to detect fatal infinite loops (cycles) that would normally crash the game engine. The program compares the number of successfully sorted plugins (processed_count) against the total number of inputted plugins (num_plugins). If the graph contained a cycle, the queue would have emptied prematurely, leaving processed_count lower than num_plugins. The system then safely traps this error, iterating through the remaining vertices to identify and print only the specific plugins whose in_degree remains greater than 0. Users then can directly resolve this conflict manually.

V. TEST CASE

A. Valid Mod List

```
printf("SCENARIO 1: Resolving a Valid Mod List\n");
int num_mods = 5;
Graph *g1 = createGraph(num_mods);

setPluginName(g1, 0, "Skyrim.esm");
setPluginName(g1, 1, "Update.esm");
setPluginName(g1, 2, "WeatherOverhaul.esp");
setPluginName(g1, 3, "WaterTextures.esp");
setPluginName(g1, 4, "WeatherPatch.esp");

addDependency(g1, 0, 1);
addDependency(g1, 1, 2);
addDependency(g1, 1, 3);
addDependency(g1, 2, 4);
addDependency(g1, 3, 4);

calculateLoadOrder(g1);
```

Fig. 16. Valid Mod List
(Source: Author)

As you can see from the Figure 16 above, this test scenario consists of 5 mods. Each mod has a distinct name and vertex ID. In this scenario there is no cycle formed within the graph, so the program will result in working load order based on mods dependencies.

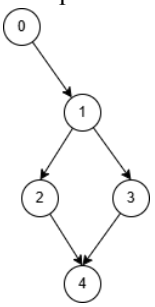


Fig. 17. Graph Model of Scenario 1
(Source: Author)

Based on the graph on Figure 17, we can see that Skyrim.esm (ID 0) doesn't have any dependency to other mods. This means Skyrim.esm have an in-degree of 0 and thus push into the queue, then into final sorted_order. Because it pushed first, Skyrim.esm will also be first in the load order.

Then the process continues by deleting each plugin's outgoing edges until their neighbor's in-degree becomes 0 and the plugins push it into the queue. Because there is no cycle formed in this mod list, after all nodes push into the final sorted_list, there is no edge left and thus a valid load order is generated.

```
SCENARIO 1: Resolving a Valid Mod List

SUCCESS! Valid Load Order Generated:
-------------------------------------
01. Skyrim.esm
02. Update.esm
03. WaterTextures.esp
04. WeatherOverhaul.esp
05. WeatherPatch.esp
-------------------------------------
```

Fig. 18. Scenario 1 Result
(Source: Author)

B. Invalid Mod List

```
printf("\nSCENARIO 2: Invalid Mods List (Infinite Loop)\n");
int num_mods = 3;
Graph *g2 = createGraph(num_mods);

setPluginName(g2, 0, "CombatOverhaul.esp");
setPluginName(g2, 1, "AnimationAddon.esp");
setPluginName(g2, 2, "WeaponsPack.esp");

addDependency(g2, 0, 1);
addDependency(g2, 1, 2);
addDependency(g2, 2, 0);

calculateLoadOrder(g2);
```

Fig. 19. Invalid Mod List
(Source: Author)

In this scenario, the number of mods is 3. Like the previous scenario, each mod has a distinct name and vertex ID. However if we look at the dependency, we can see that there is a cycle form within the graph (0 → 1 → 2 → 0).

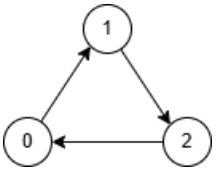


Fig. 20. Graph Model of Scenario 2
(Source: Author)

Because there is a cycle form within the graph, topological sorting is impossible. This will result in invalid load order. Then the program will isolate the error so the users can fix it manually.

```
SCENARIO 2: Invalid Mods List (Infinite Loop)

ERROR! Circular Dependency Detected!
The graph is not a valid DAG. A topological sort is impossible.
The following plugins are caught in a circular loading loop:
 - CombatOverhaul.esp
 - AnimationAddon.esp
 - WeaponsPack.esp
```

Fig. 21. Scenario 2 Result
(Source: Author)

C. Rule of One Conflict

```
printf("\nSCENARIO 3: Managing the 'Rule of One' via a Resolution Patch\n");
int num_mods = 4;
Graph *g3 = createGraph(num_mods);

setPluginName(g3, 0, "Skyrim.esm");
setPluginName(g3, 1, "WeaponDamageOverhaul.esp");
setPluginName(g3, 2, "WeaponMeshReplacer.esp");
setPluginName(g3, 3, "Weapon_ConsistencyPatch.esp");

addDependency(g3, 0, 1);
addDependency(g3, 0, 2);
addDependency(g3, 1, 3);
addDependency(g3, 2, 3);

calculateLoadOrder(g3);
```

Fig. 22. Rule of One
(Source: Author)

Scenario 3 demonstrates how the DAG model actively manages the Creation Engine's hardcoded 'Rule of One'. In this scenario, two independent plugins (WeaponDamageOverhaul.esp and WeaponMeshReplacer.esp) attempt to overwrite the identical base game record for an 'Iron Sword'. Because the engine cannot merge these files at runtime, loading them sequentially without intervention will result in one mod's features being entirely erased by the other. To resolve this, the user creates a new plugin, Weapon_ConsistencyPatch.esp, that manually merges the data.

For the patch to successfully overwrite the previous two mods, the topological sort must guarantee it is placed at the bottom of the local load sequence. By defining directed edges from both the Damage Overhaul and the Mesh Replacer pointing toward the Patch (1 → 3 and 2 → 3), the Patch is assigned an initial in-degree of 2.When Kahn's algorithm executes, the Patch is withheld from the queue until both conflicting mods have been safely processed and appended to the load order. The output successfully places the Patch last. Consequently, when this generated sequence is injected into the game engine, the engine's native Rule of One behaves exactly as intended, accepting the merged data from the final plugin.

```
SCENARIO 3: Managing the 'Rule of One' via a Resolution Patch

SUCCESS! Valid Load Order Generated:
-------------------------------------
01. Skyrim.esm
02. WeaponMeshReplacer.esp
03. WeaponDamageOverhaul.esp
04. Weapon_ConsistencyPatch.esp
-------------------------------------
```

Fig. 23. Scenario 3 Result
(Source: Author)

VI. CONCLUSION

This paper proves that by abstracting a modding environment into a directed acyclic graph (DAG), we successfully mapped discrete plugins as vertices and their requisite dependencies as directed edges. This allows the use of Kahn's Algorithm to create working load order and isolate the conflicting plugins within the modding environment. With this algorithm a fail-safe against human logic errors can be provided when modding with a large environment that consists of hundreds of mods. This will make modding Skyrim become easier and less prone to errors.

ATTACHMENTS

Presentation Video:
https://youtu.be/iFHWkJ-rzUU

Code Implementation:
https://github.com/FAIshalAN/Makalah-Matdis

ACKNOWLEDGMENT

First of all, I would like to express my deepest gratitude to Allah SWT, for it is by His grace that I have been able to successfully complete this paper. I would also

like to express my deepest gratitude to the lecturer of IF1220 Discrete Mathematics, Dr. Ir. Rinaldi Munir, MT., who has guided me throughout this course from beginning to end. It is thanks to his teaching that this paper was possible. I would also like to express my gratitude to the assistants who have helped us throughout the discrete mathematics course. I would also like to thank my friends and everyone who has assisted me during the process of preparing this paper.

I hope this paper will not only serve as an assessment component for me, but will also be beneficial to the community and future research. May this paper serve as a guide and source of ideas for future work.

REFERENCES

[1] E. W. Weisstein, "Graph," MathWorld--A Wolfram Web Resource. https://mathworld.wolfram.com/Graph.html (accessed Jun. 17, 2026).

[2] R. Munir, "Graf (Bagian 1)," Program Studi Teknik Informatika, Institut Teknologi Bandung, 2024. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

[3] GeeksforGeeks, "Introduction to Directed Acyclic Graph," GeeksforGeeks. https://www.geeksforgeeks.org/dsa/introduction-to-directed-acyclic-graph/ (accessed Jun. 18, 2026).

[4] E. Fanning, "Game Mods," Academia.edu. https://www.academia.edu/48437005/Game_Mods (accessed Jun. 18, 2026).

[5] Bethesda Game Studios, "Creation Kit Wiki," Creation Kit. https://www.creationkit.com/ (accessed Jun. 18, 2026).

[6] STEP Modifications, "SkyrimSE:2.3," STEP Modifications Wiki. https://stepmodifications.org/wiki/SkyrimSE:2.3 (accessed Jun. 18, 2026).

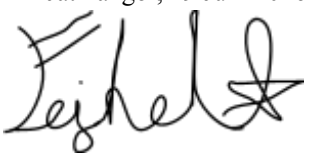
[7] Modding.wiki, "Creating Mods," Modding.wiki. https://modding.wiki/en/stalker2heartofchornobyl/developers (accessed Jun. 18, 2026).

[8] GeeksforGeeks, "Topological sorting (indegree based solution)," GeeksforGeeks. https://www.geeksforgeeks.org/dsa/topological-sorting-indegree-based-solution/ (accessed Jun. 18, 2026).

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Jatinangor, 19 Juni 2026



Faishal Ahmad Nurdin - 13525027